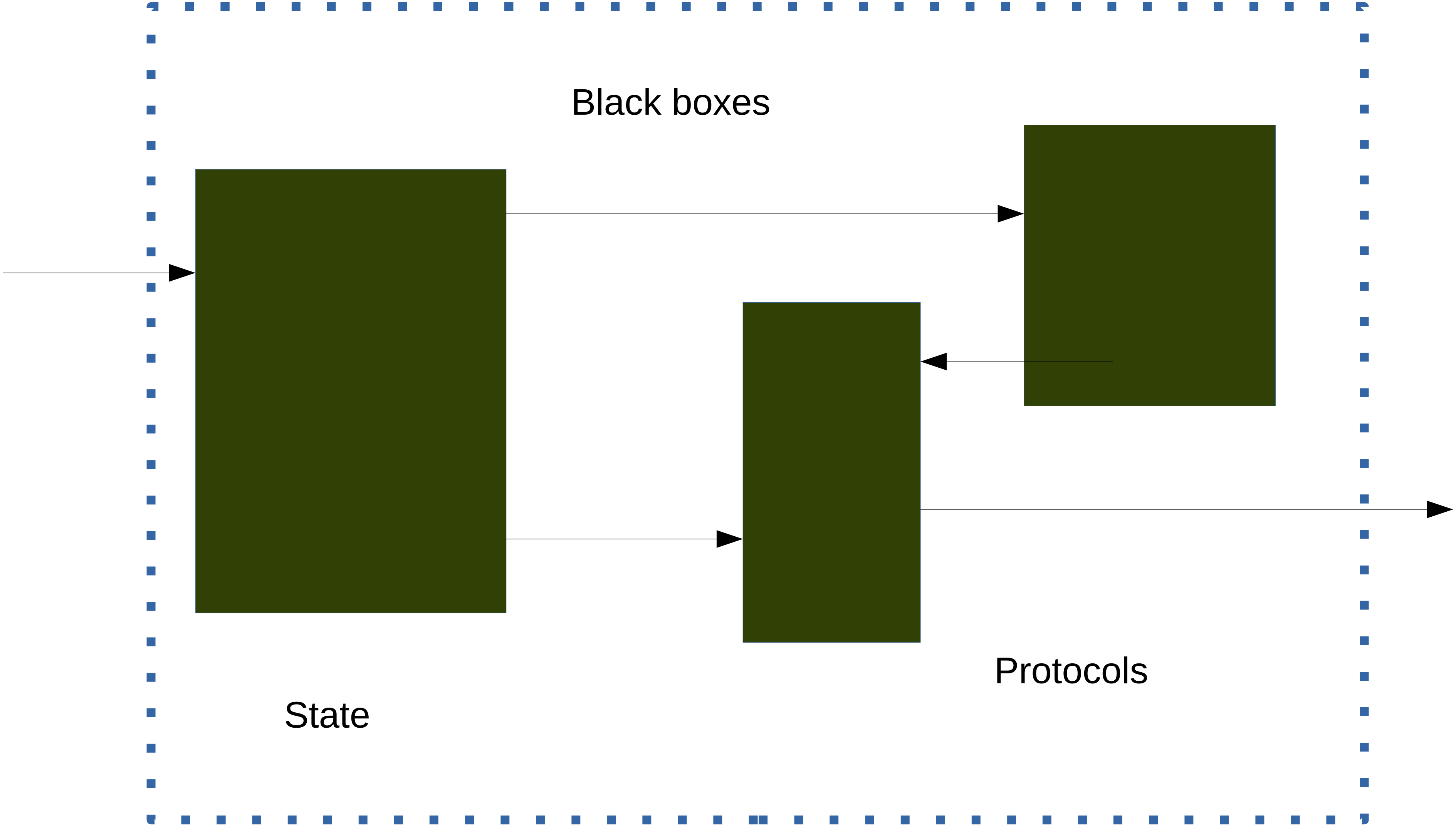
How do we
make software
that runs
forever and
has no faults?

Programming
is about
connecting
things
together

The Big picture



Black boxes

Protocols

State

Abstraction

- Systems are made from black boxes
- Two systems are the same if they behave the same way *“observational equivalence”*
- Interactions can be defined formally in protocols
- Configurations can be defined formally
- Protocols and Configurations can be described by content hashes
- Systems have state
- State can be described with content hashes

Monads Pipes
Plumbing
Middle Men
And all that jazz ..

The Important bits

- Composability
- Contract checking
- Black boxes, pipes, protocols
- Content hashes

If I get off topic or am running out of time tell me ...

Pipes were a great idea but what comes next?

Joe Armstrong

Two ways to connect things together

- Link together in memory

  “shared memory concurrency”

- Send messages

  “Message passing concurrency”

Shared memory

+ Efficient

- Locks

- Fault intolerant

- No concurrency control

- non-scalable

- tangles things together

- version nightmares

Message passing

+ Fault isolation

+ Scalable

+ Late binding

+ Version Bliss

+ Contracts

+ The “core” of OOP

It's all about
Composing Computations

Why?

Make re-usable things that can be
re-used in all contexts

Plan

- Compute sin(2x) 36 – slides

  Visiting Monads, Pipes, Debugging, Conceptual Integrity, Proofs, Theorems, the Curry-Howard Correspondence, and the Higgs Boson
- Pipes
- Contracts
- Heaven Purgatory and Hell

Compute

sin(2*X)

With debugging code

```
sin(X) → math:sin(X).

square(X) → X*X.

sinSquare(X) →
    sin(square(X)).
```

f(g(X)) F and G are composable

g(f(X))

In Erlang you can always compose functions

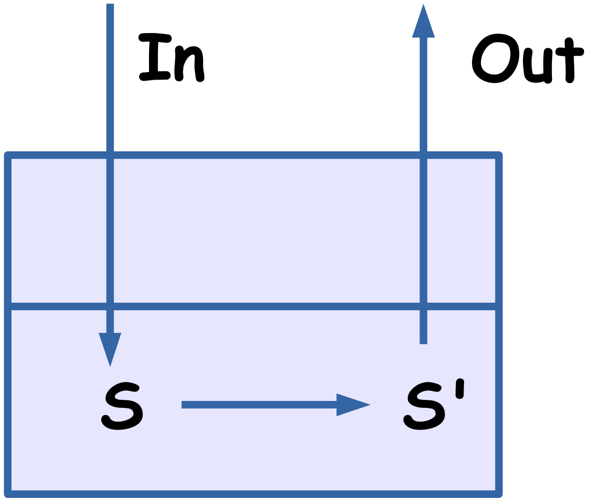
Do you want type errors to occur at compile or run time?

In Haskell/Java/C?? you cannot compile these if the type system complains

Hidden State Prevents Composability

Hidden State = Side Effect

Referential transparency



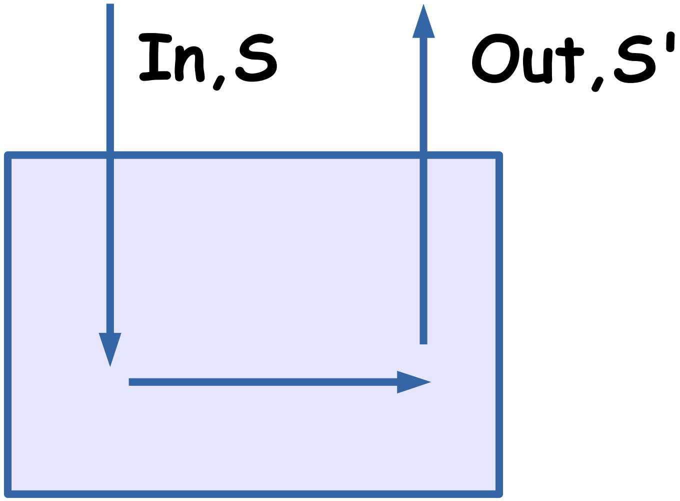
In

Out

S

S'

OOP

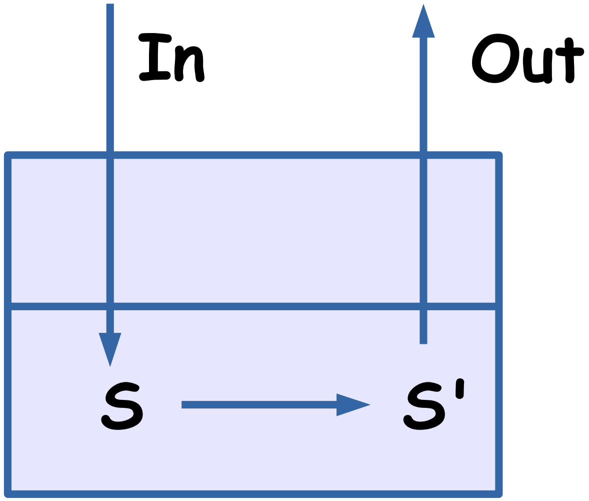


In,S

Out,S'

FP

OOP



OOP

OOPLs make a religion
of hiding data inside the object
this makes it very difficult to
reason about the behaviour
of the object.

OOPLs have no theoretical
basis

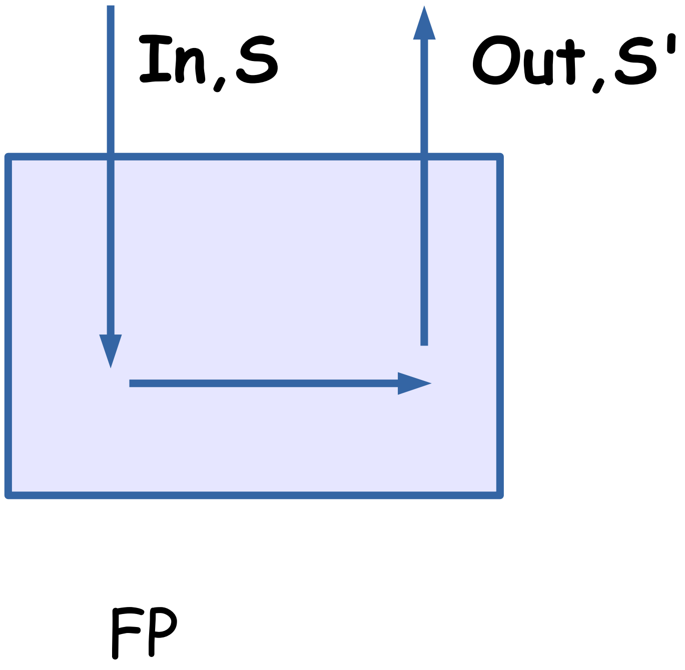
OOP is the art of hiding side effects

Functional programming languages

FLPs carry state with them wherever the flow of control goes. Different FPLs provide different notations and mechanisms for hiding this from the user.

In Erlang we hide the state in a process. In Haskell in a monad

FLPs have are based on a formal mathematical model
Lambda calculus (Pi calc, CSP)



Carrying state in and out of every function is inconvenient – how can we hide this?

Monads

In functional programming, a monad is a structure that represents computations defined as sequences of steps: a type with a monad structure defines what it means to chain operations, or nest functions of that type together. This allows the programmer to build pipelines that process data in steps, in which each action is decorated with additional processing rules provided by the monad.[1] As such, monads have been described as "programmable semicolons"; a semicolon is the operator used to chain together individual statements in many imperative programming languages

From: wikipedia

```
sin(X) ->  math:sin(X).
square(X) -> X*X.

sin_square1(X) -> sin(square(X)).

%% > monads:sin_square1(3).
%% 0.4121184852417566

compose(F, G) ->
    fun(X) -> F(G(X)) end.

sin_square2(X) -> (compose(fun sin/1, fun square/1))
(X).

%% > monads:sin_square2(3).
%% 0.4121184852417566
```

These are in the modulemonads.erl show if you have time

Maths:

SinSquare = sin ∘ square

No side effects so debug string must be an output of the function

```
sin_d(X)     -> {math:sin(X),"sine called"}.
square_d(X) -> {X*X, "square called"}.

sin_square3(X) ->
    (compose(fun sin_d/1, fun square_d/1))(X).

%% > monads:sin_square3(3).
%% ** exception error: bad argument
%%      in function  math:sin/1
%%         called as math:sin({9,"square called"})
%%      in call from monads:sin_d/1 (monads.erl, line 7)
```

We lost composability

Hop over the next few slides if running out of time

What's wrong?

```
sin_d(X)     -> {math:sin(X),"sine called"}
```

Is wrong we'd like it to be

```
sin_d(X, S1) → {math:sin(X), S1 ++ “sine called”}
```

WHY?

So we can make a pipeline

sin(square(X))

NumberIn | square | sin | NumberOut

The data flowing over the boundary in always of type Number

Number → Number → Number → Number

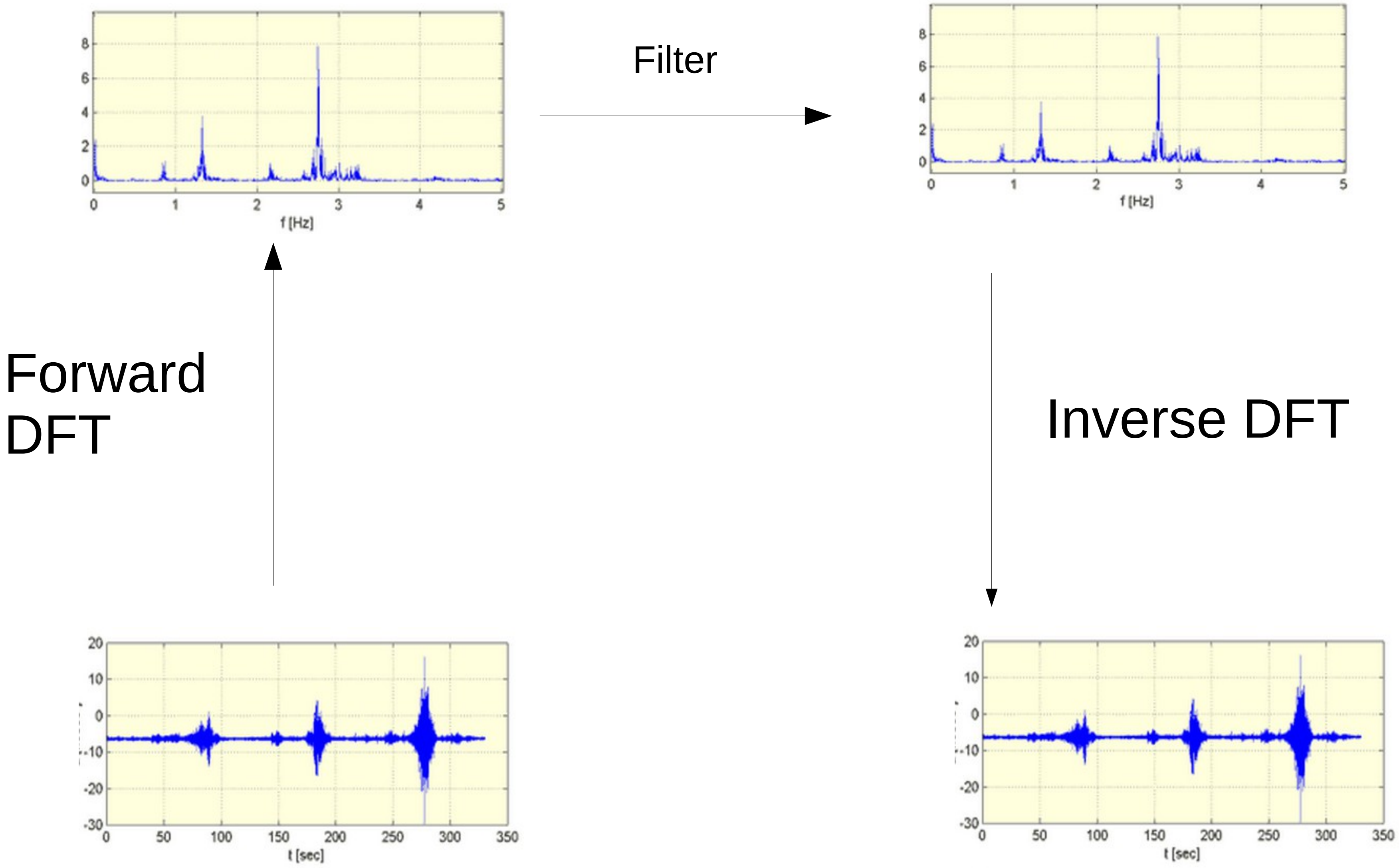
So we can write f(g(h(i(X))))

sin_d(square_d(X))

{Number, String1} → {Number2, String2}
→ {Number3, String3}

```
bind(F) ->
    fun({X,Str}) ->
        {R,Str1} = F(X),
        {R, Str ++ ";" ++ Str1}
    end.

sin_square4(X) ->
    (compose(bind(fun sin_d/1), bind(fun square_d/1))({X,""}).

%% > monads:sin_square4(3).
%% {0.4121184852417566,";square called;sine called"}

%% Hooray we got back composability
```

$$\begin{tikzcd}
X \arrow[r] & Y \arrow[d, "unit"] \\
A \arrow[u, "bind"] & B
\end{tikzcd}$$



Filter

Forward DFT

Inverse DFT

Meanwhile in Erlang ...

F(G(X)) is used for small steps

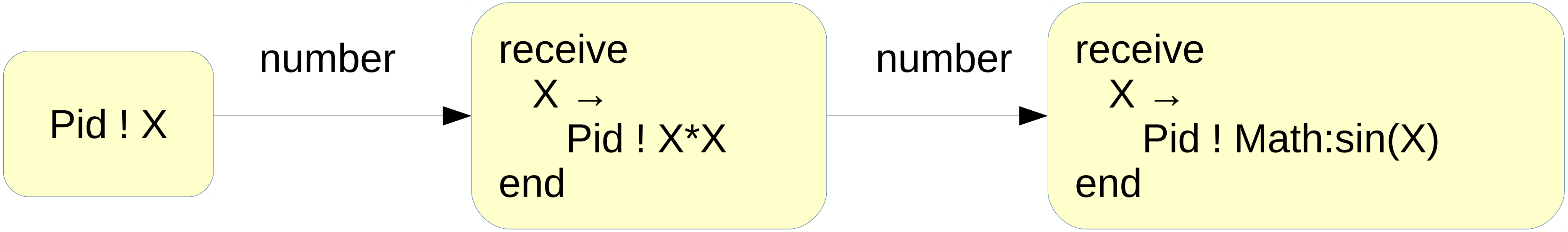
Pipes are used for big steps

Input | G | F | Output

```
find *erl | grep “fred” | uniq | wc
```

Note the automatic parallelism

But we don't call them pipes we call them processes



```
Pid ! X
```

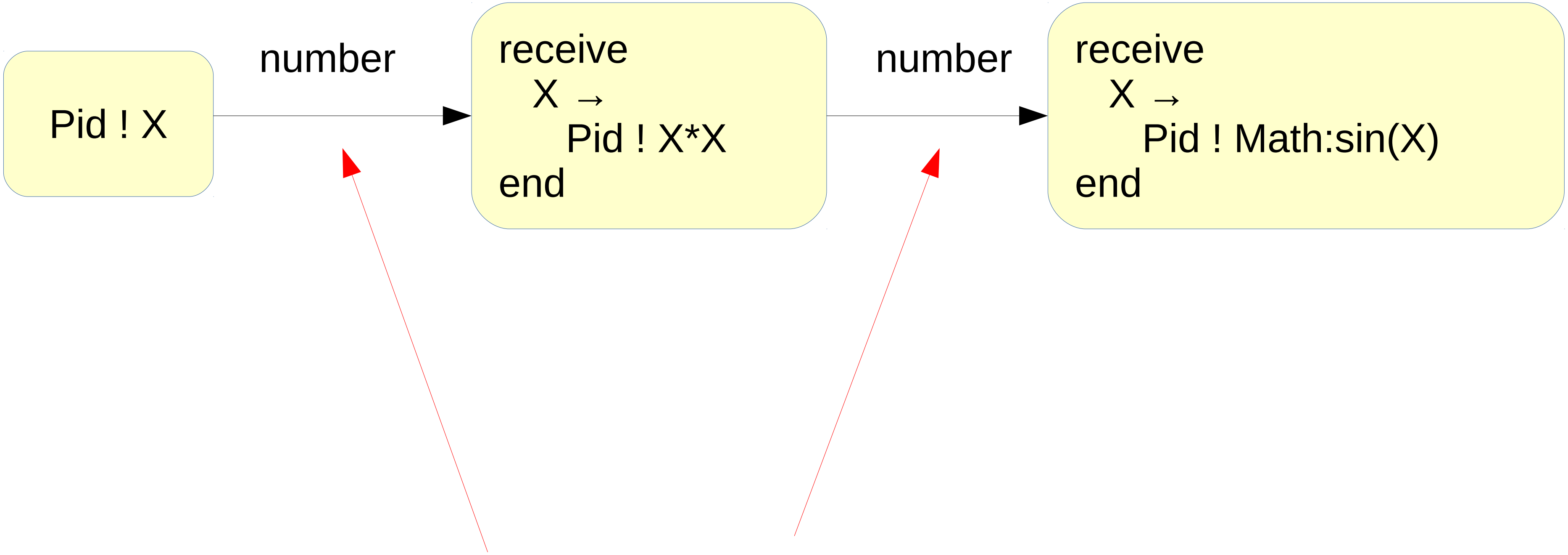
number

```
receive
   X →
      Pid ! X*X
end
```

number

```
receive
   X →
      Pid ! Math:sin(X)
end
```

How do we add debugging?



```
Pid ! X
```

number

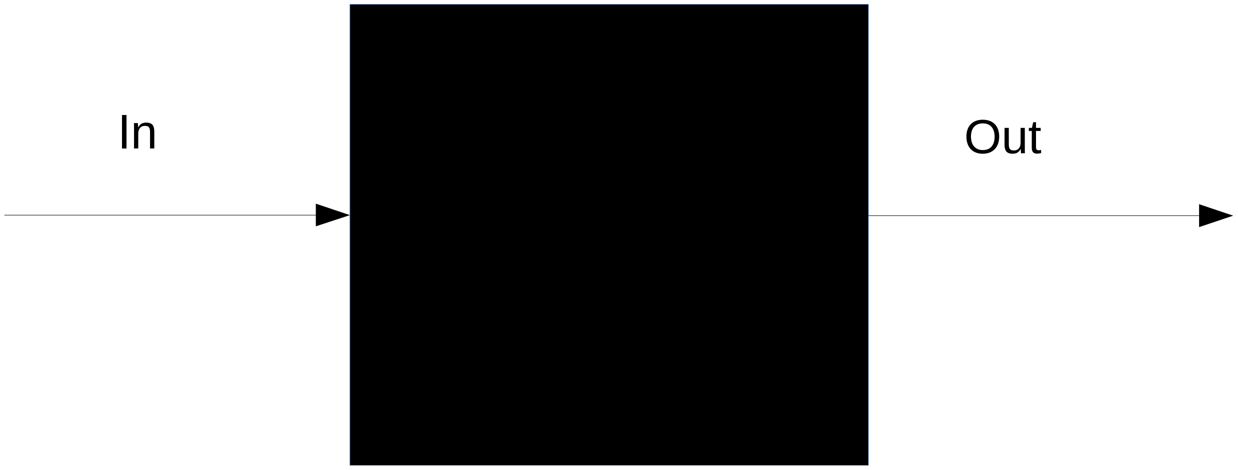
```
receive
  X →
    Pid ! X*X
end
```

number

```
receive
  X →
    Pid ! Math:sin(X)
end
```

Spy on the communication channels (like wireshark)

Observational Equivalence



Two systems are equivalent if we cannot distinguish them
By observing any differences in their input/output behavior

Remember

Small steps = function calls
Big steps = processes

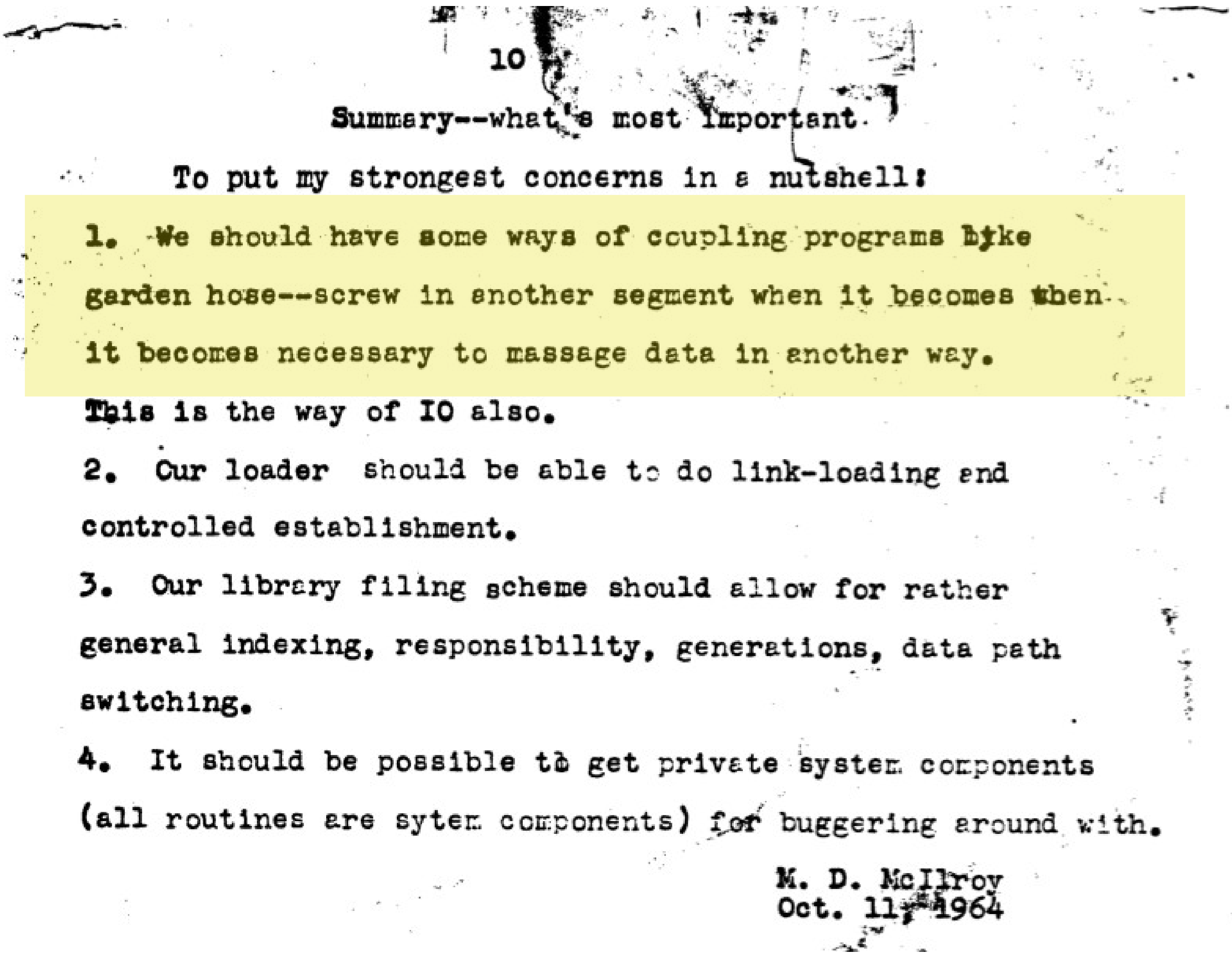
And ...

Functions calls run sequentially

Processes run in parallel

So we have a nice way to think
About parallel algorithms

Pipes



Summary--what's most important.

To put my strongest concerns in a nutshell:

1. We should have some ways of coupling programs like garden hose--screw in another segment when it becomes then it becomes necessary to massage data in another way. This is the way of IO also.

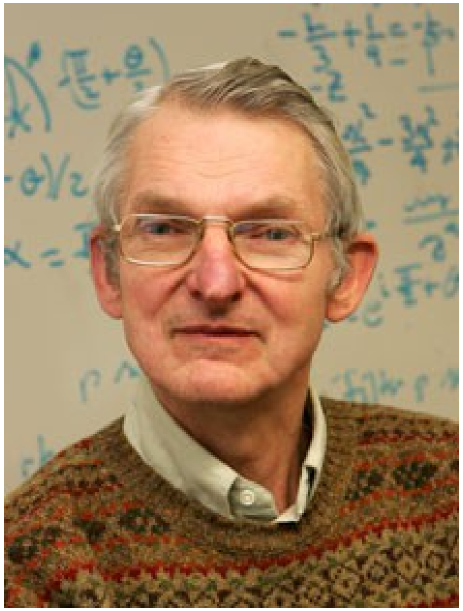
2. Our loader should be able to do link-loading and controlled establishment.

3. Our library filing scheme should allow for rather general indexing, responsibility, generations, data path switching.

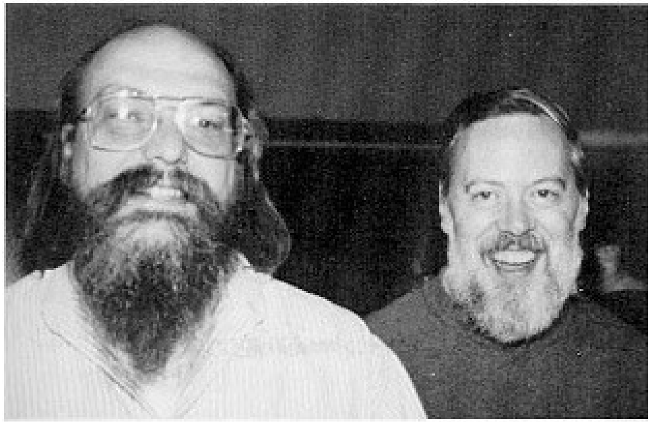
4. It should be possible to get private system components (all routines are syter components) for buggering around with.

M. D. McIlroy
Oct. 11, 1964

“.. he conceived Unix pipes, which allow programs to work together with no knowledge of each other...”



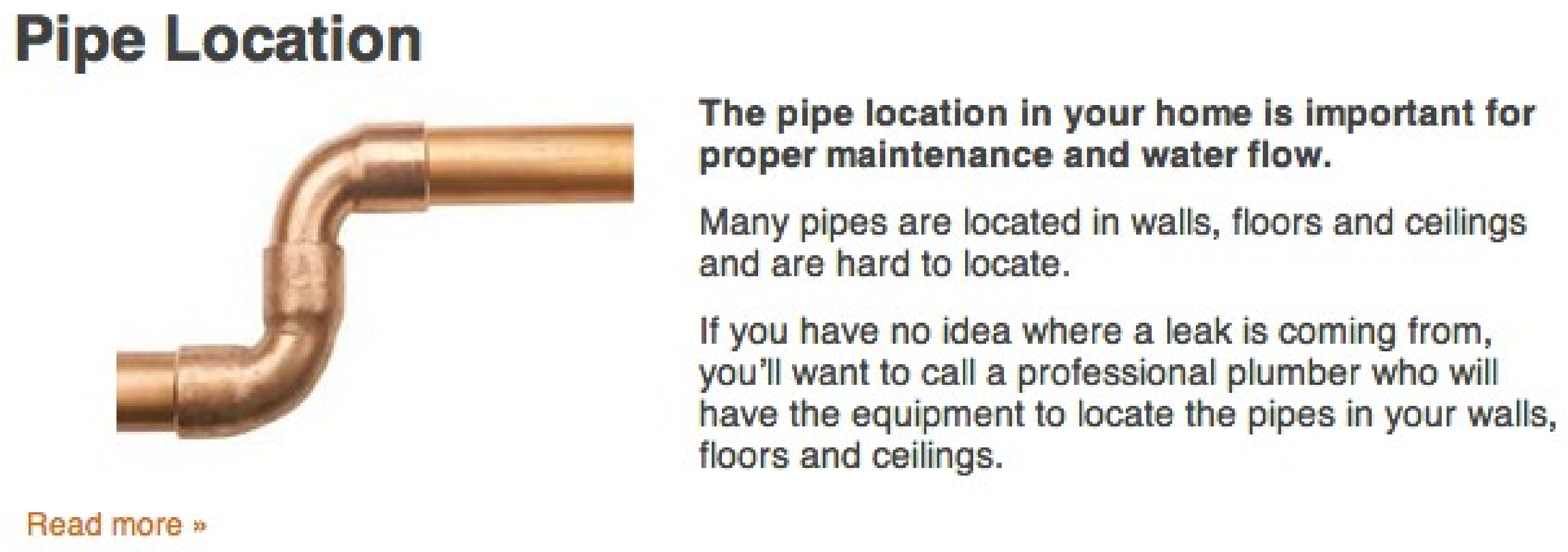
M.Douglas McIlroy



“Doug has been explicit in saying that he very nearly exercised managerial control to get pipes installed.”

“Point 1's garden hose connection analogy, though, is the one that ultimately whacked us on the head to best effect.”

http://cm.bell-labs.com/cm/cs/who/dmr/mdmpipe.html



Pipe Location

The pipe location in your home is important for proper maintenance and water flow.

Many pipes are located in walls, floors and ceilings and are hard to locate.

If you have no idea where a leak is coming from, you'll want to call a professional plumber who will have the equipment to locate the pipes in your walls, floors and ceilings.

Read more »

http://www.elocalplumbers.com/content/plumbing-articles/pipes

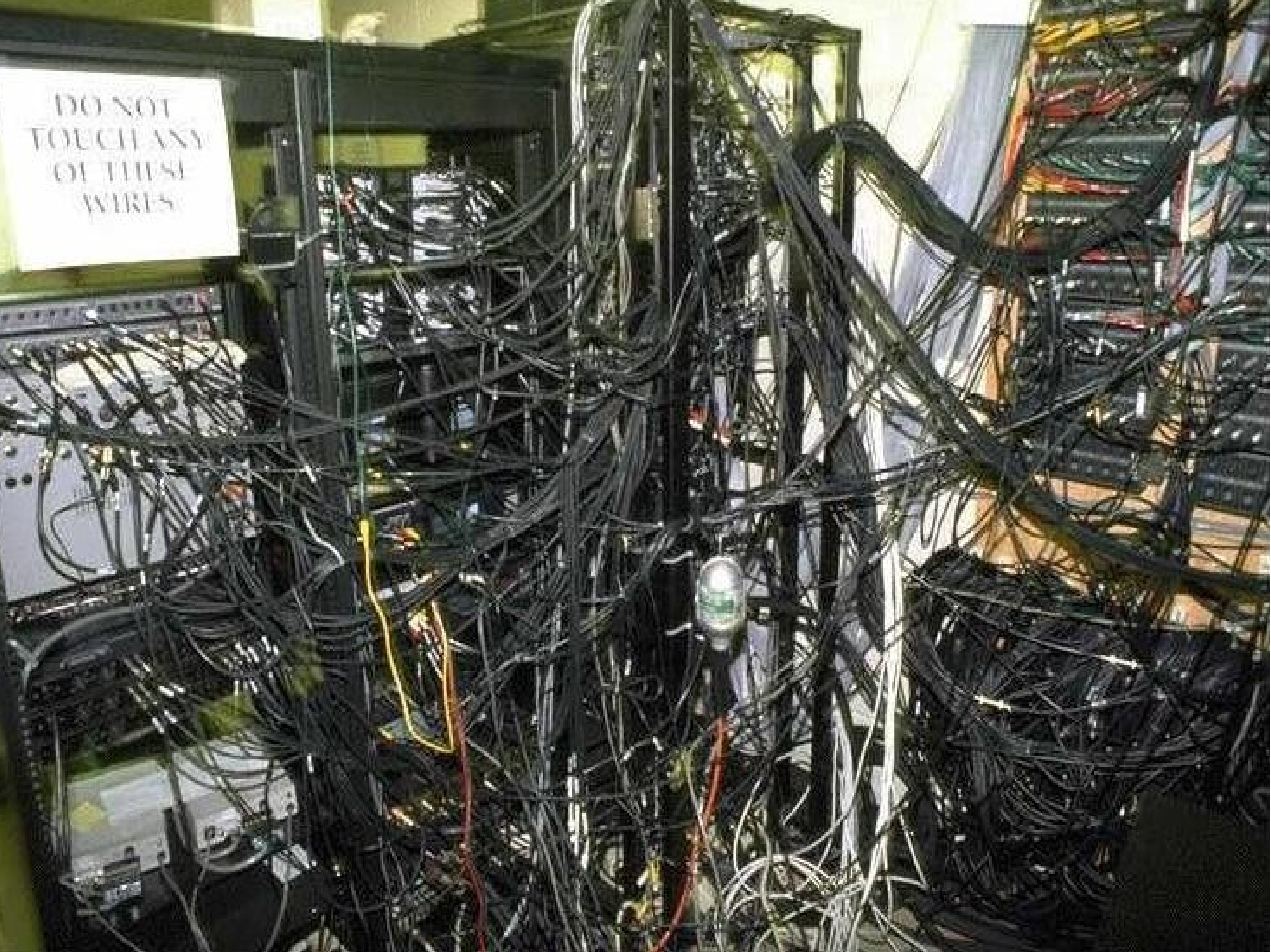
Handyman tips

1. When you move into a new house or property try to locate the main stopcock which shuts off the water supply to the house-do not wait until you have a major problem, then it will be too late

2. Fit service valves to all your pipes ,this will allow you to work on the bathroom sink for instance without it affecting the water supply in the bath, shower or toilet for instance, this gives every item its own identity and allows you to change taps or solve water leak problems without shutting off the water supply in the entire house.

3. ...

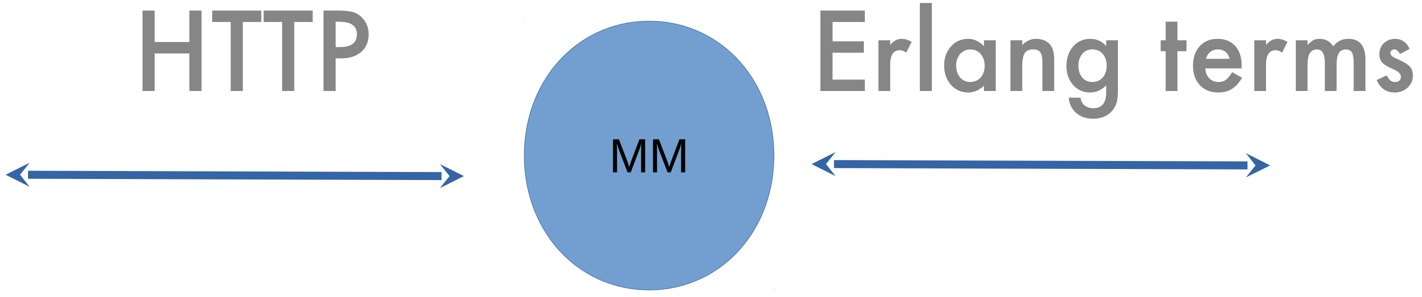
http://www.handymanlosangeles.us/tips_articles/10_most_common_plumbing_problems.html


DO NOT
TOUCH ANY
OF THESE
WIRES

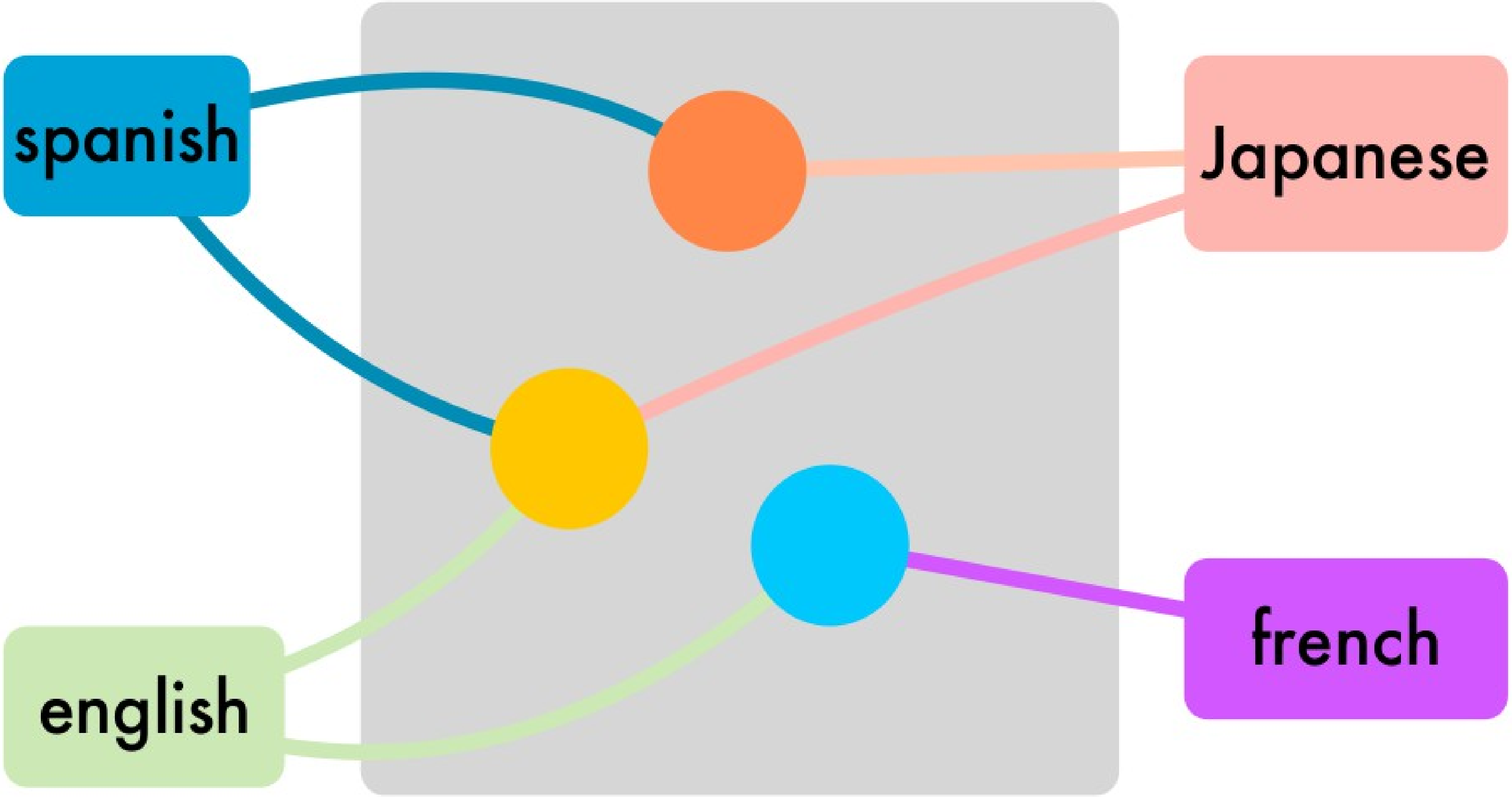
http://www.elocalplumbers.com/content/plumbing-articles/residential-boilers-troubleshooting-3032

Middle Men

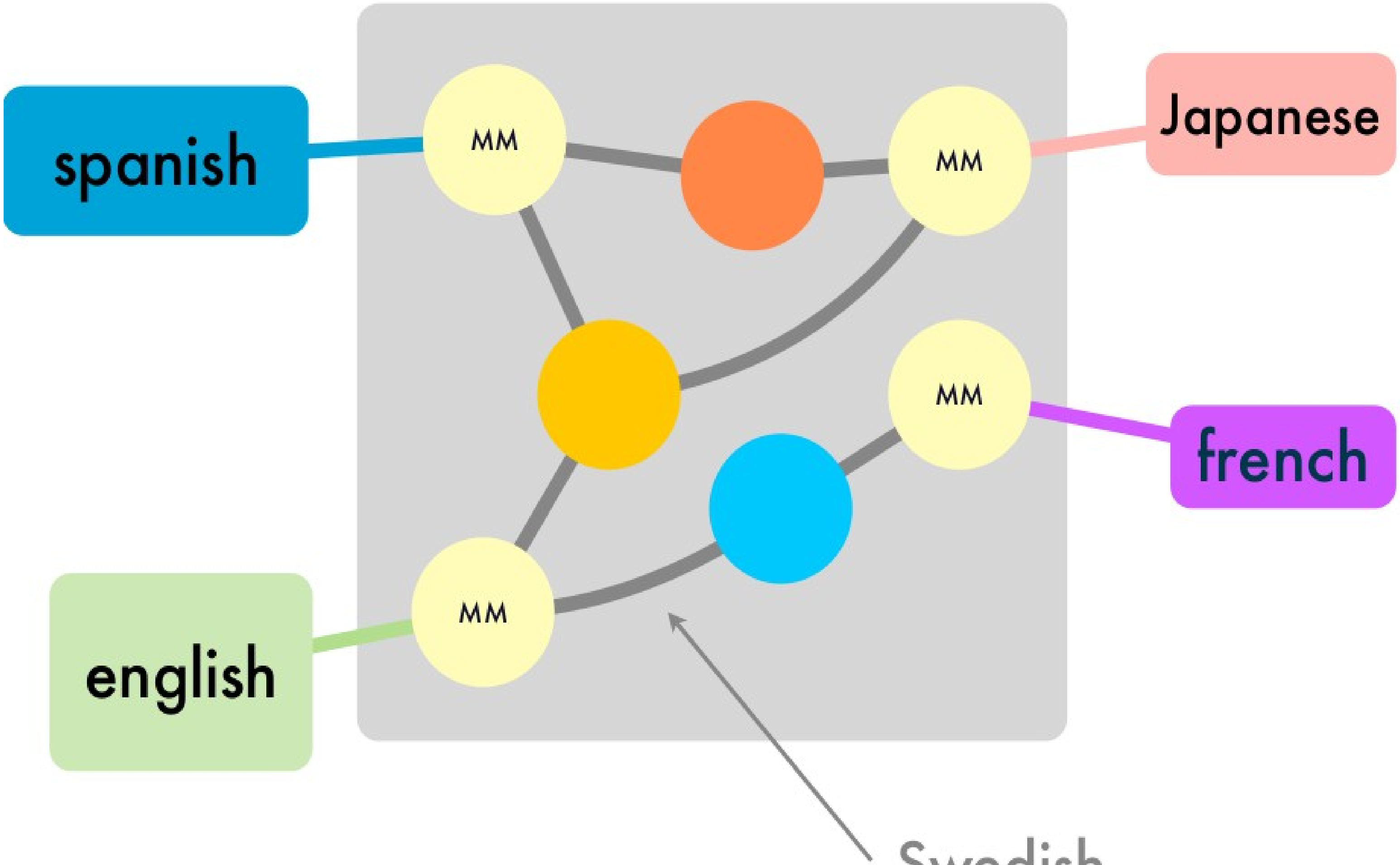
The Middle Man



The middle man creates the illusion that the external world is composed of Erlang processes



Complexity ~ $O(N^2)$

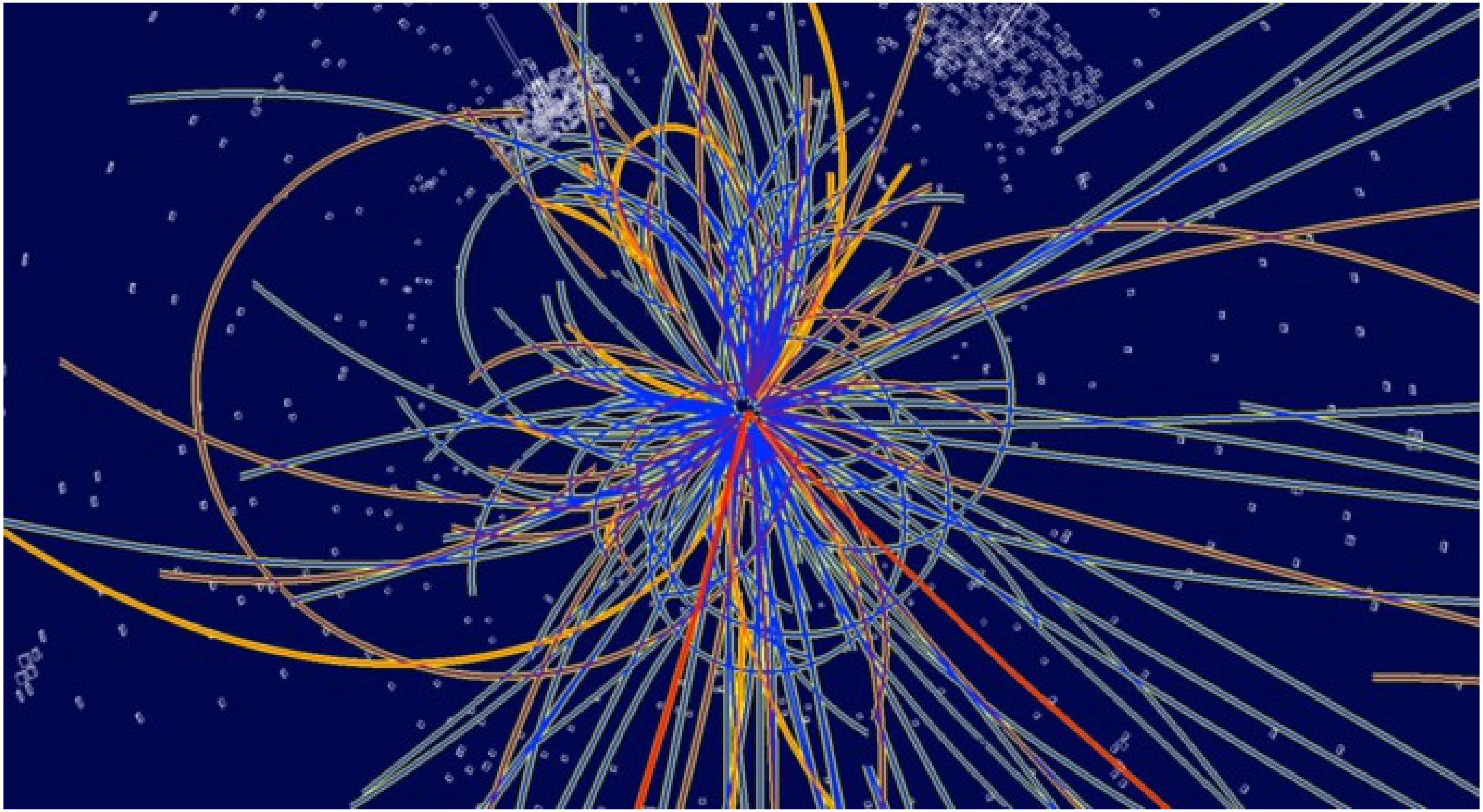


Swedish

Complexity ~ O(N)

Conceptual integrity

The MM is the bringer of order - it imposes the rule “everything in the world is an Erlang process”

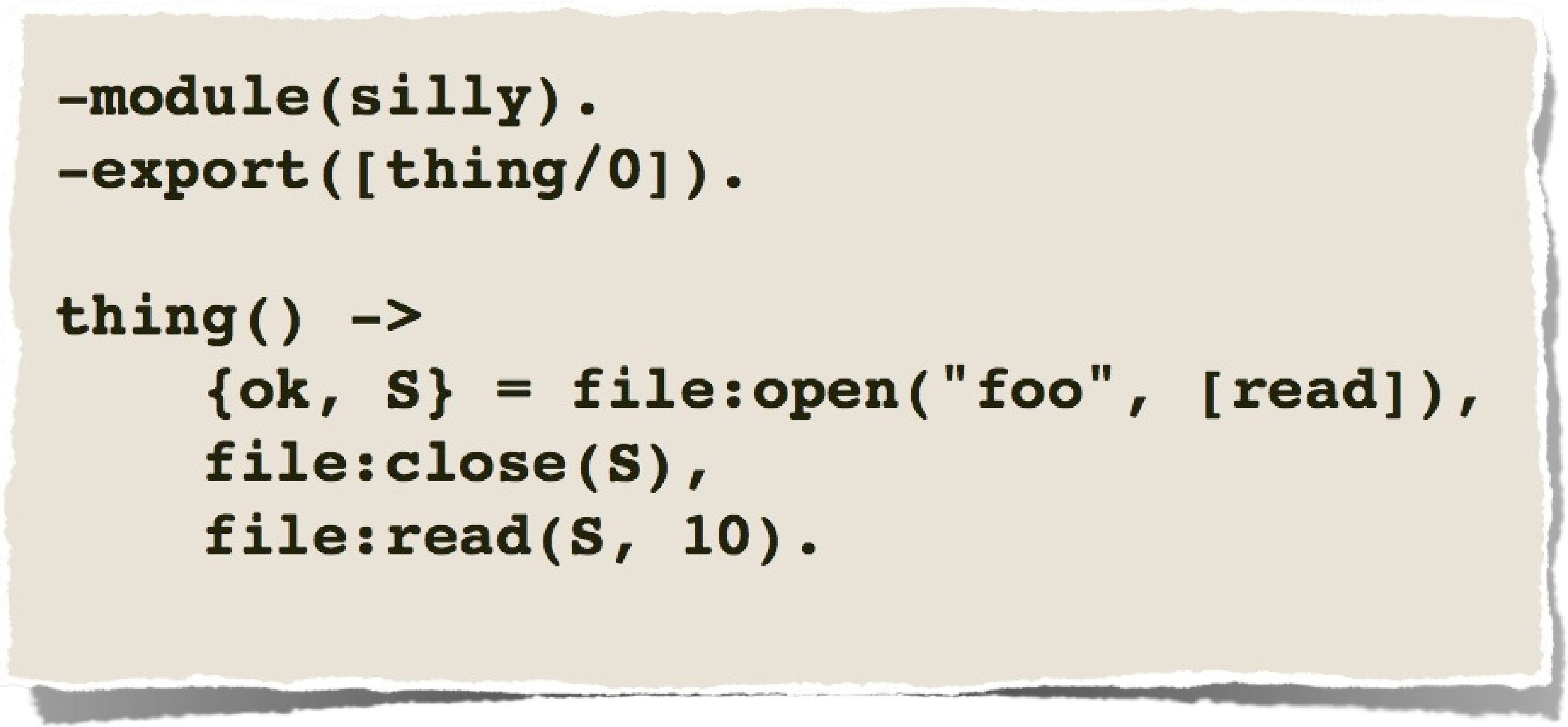


The Middle Man is the Higgs Boson of Erlang

Describing interactions

APIs
Don’t work

```
-spec open(File, Modes) ->
     {ok, Handle} | {error, Reason}.

-spec close(Handle) ->
     ok | {error,Reason}

-spec  read(Handle, Int) ->
     {ok,Data} | {error, Reason}.
```



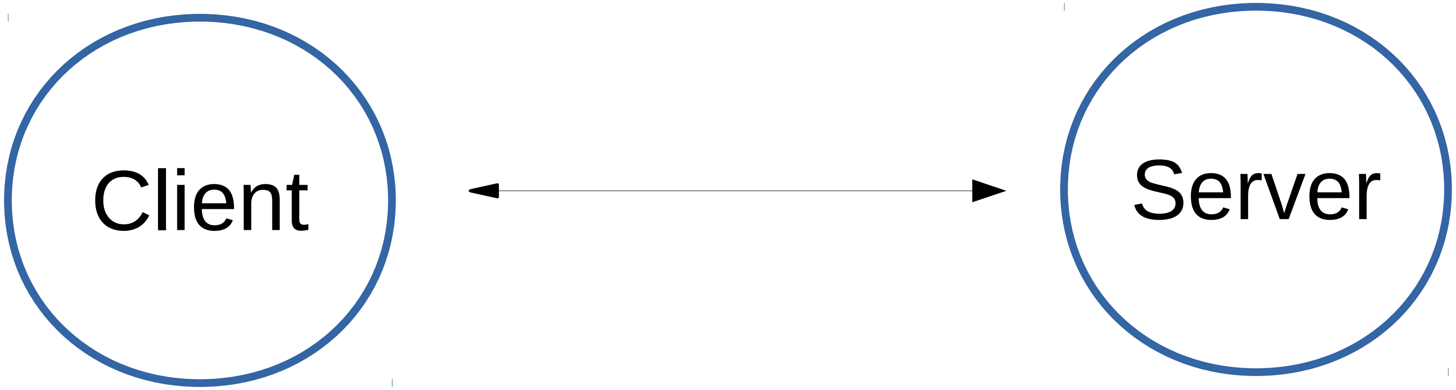
```
-module(silly).
-export([thing/0]).

thing() ->
     {ok, S} = file:open("foo", [read]),
     file:close(S),
     file:read(S, 10).
```

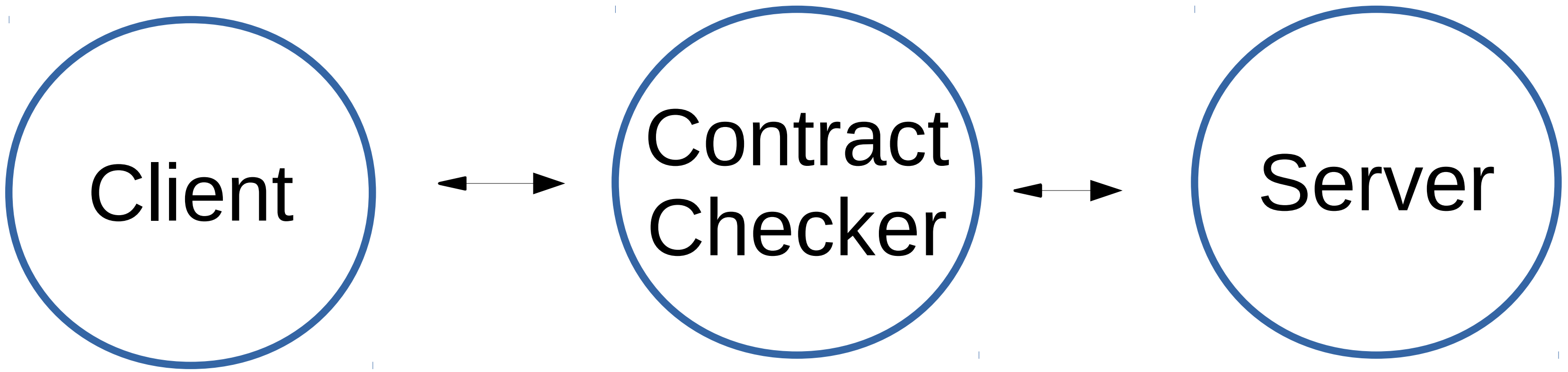
“Session types”

```
-spec start x open(File, Modes) ->
    {ok, Handle}    x ready |
    {error, Reason} x closed.

-spec ready x close(Handle) ->
     ok x closed | {error, Reason} x closed.

-spec ready x read(Handle, Int) ->
      {ok,Bin} x ready |
        {error, E} x closed.
```

Session
Type Contracts

Website login protocol

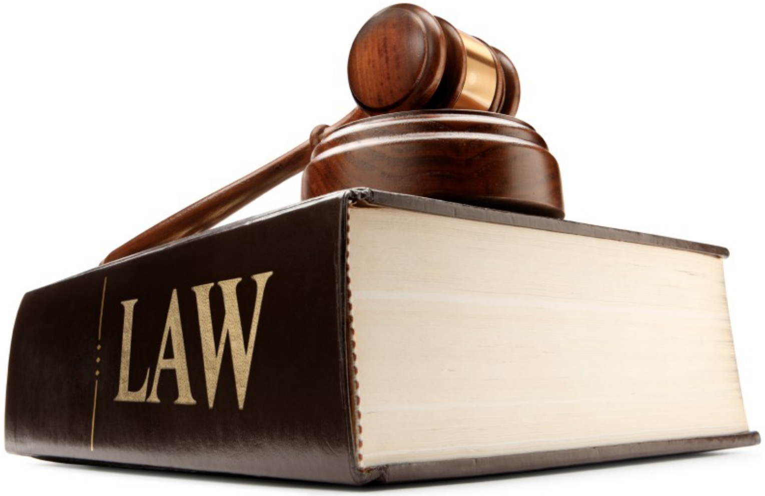

Client
Server



Client
Contract Checker
Server

Contracts are 4-tuples

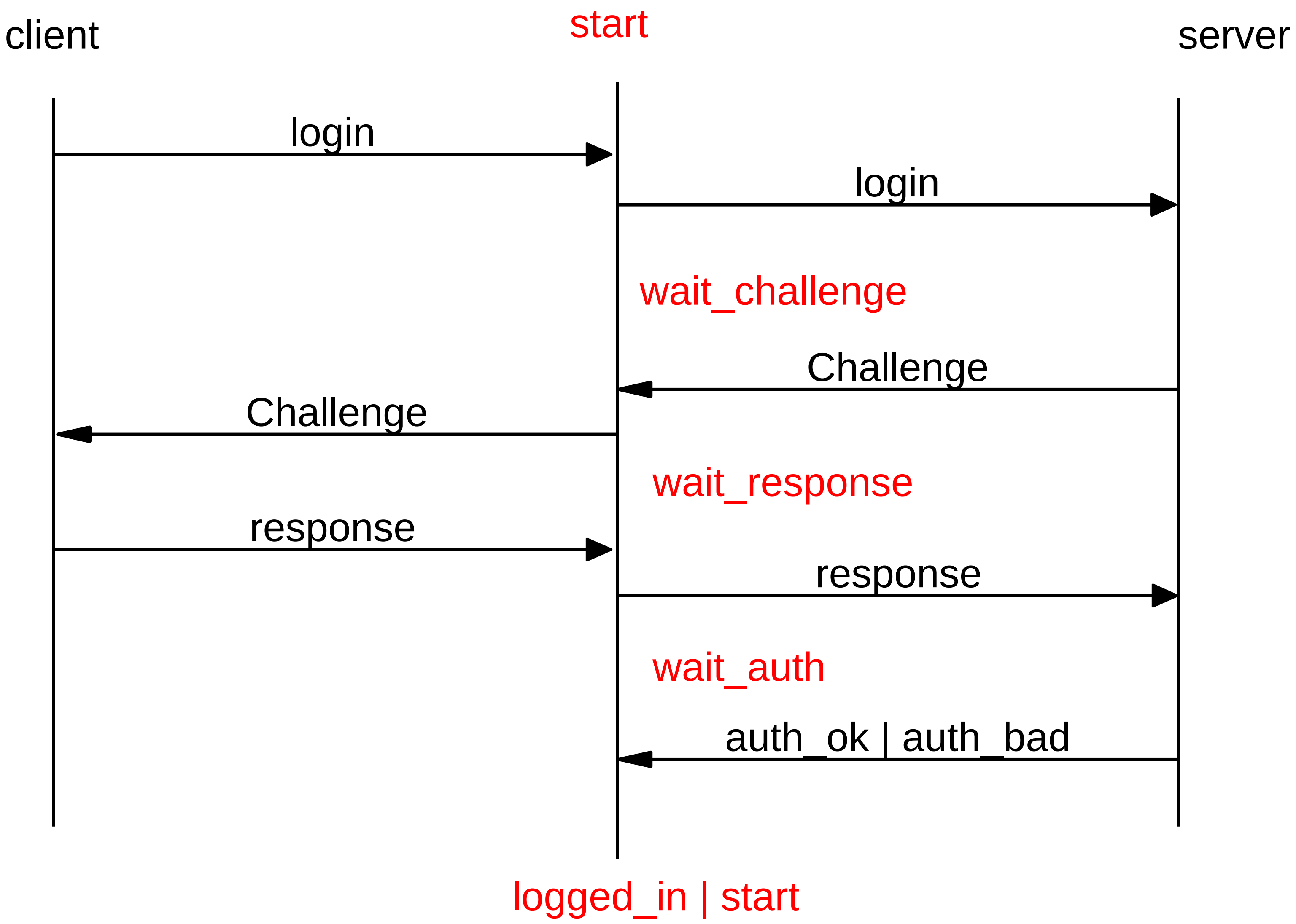
StateIn x MsgIn -> MsgOut x StateOut

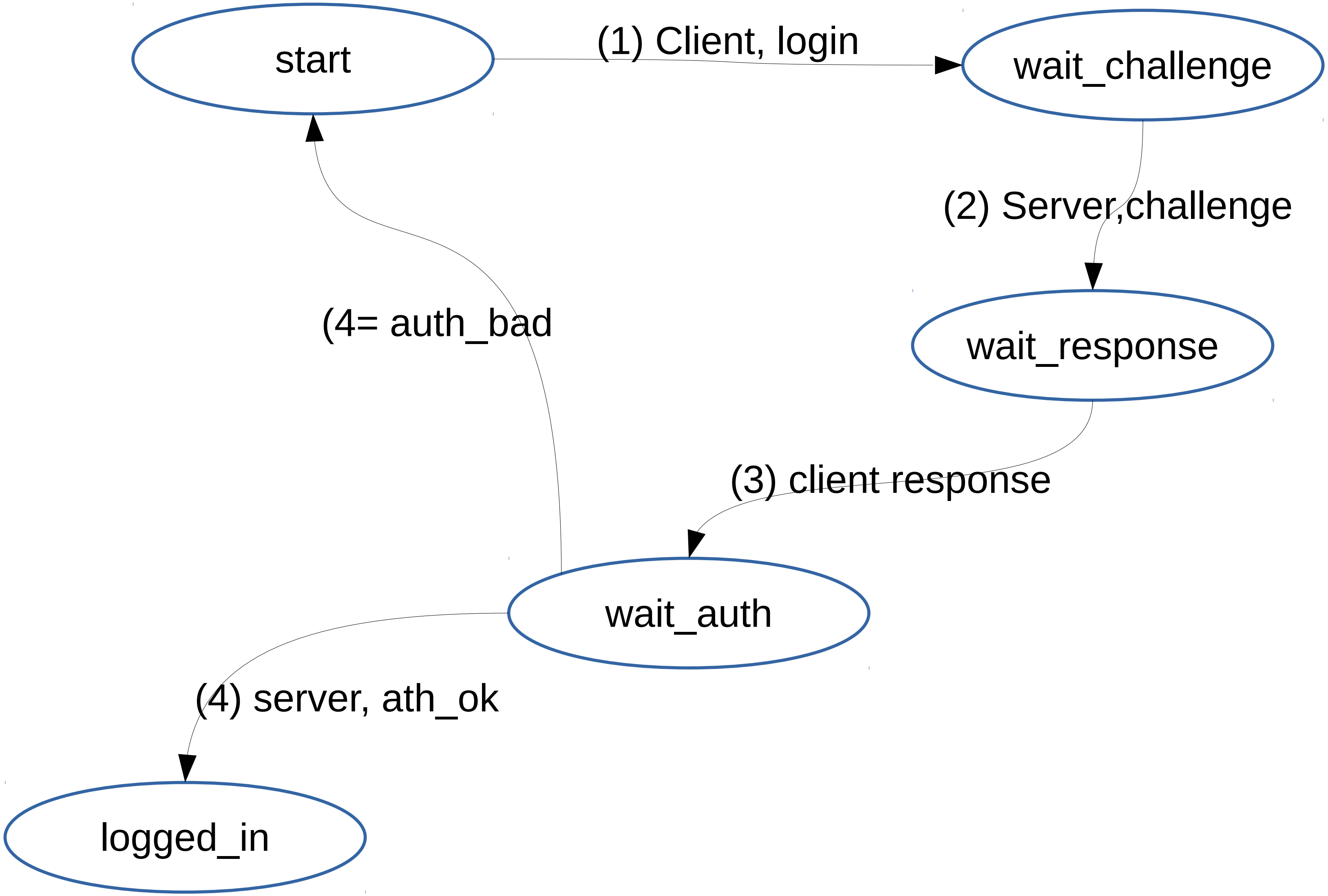


FSM in Javascript

```
var fsm = new Array();

fsm =
 ['start',          'client', 'login',     'wait_challenge'],
 ['wait_challenge','server', 'challenge', 'wait_response'],
 ['wait_response', 'client', 'response',  'wait_auth'],
 ['wait_auth',      'server', 'auth_ok',   'logged_in'],
 ['wait_auth',      'server', 'auth_bad',  'start']];
```

MSC


client
start
server
login
login
wait_challenge
Challenge
Challenge
wait_response
response
response
wait_auth
auth_ok | auth_bad
logged_in | start


start
(1) Client, login
wait_challenge
(2) Server,challenge
wait_response
(4= auth_bad
(3) client response
wait_auth
(4) server, ath_ok
logged_in

FSM in Javascript

```
var fsm = new Array();

fsm =
 ['start',           'client', 'login',      'wait_challenge'],
 ['wait_challenge','server', 'challenge', 'wait_response'],
 ['wait_response', 'client', 'response',  'wait_auth'],
 ['wait_auth',      'server', 'auth_ok',    'logged_in'],
 ['wait_auth',      'server', 'auth_bad',   'start']];
```

Messages are described by types

```
var type = new Array();

type['login']     = {name:'string'};
type['challenge'] = {ran:'string'};
type['response']  = {token:'string'};
type['auth_ok']   = {};
type['auth_bad']  = {};
```

Instance of login type

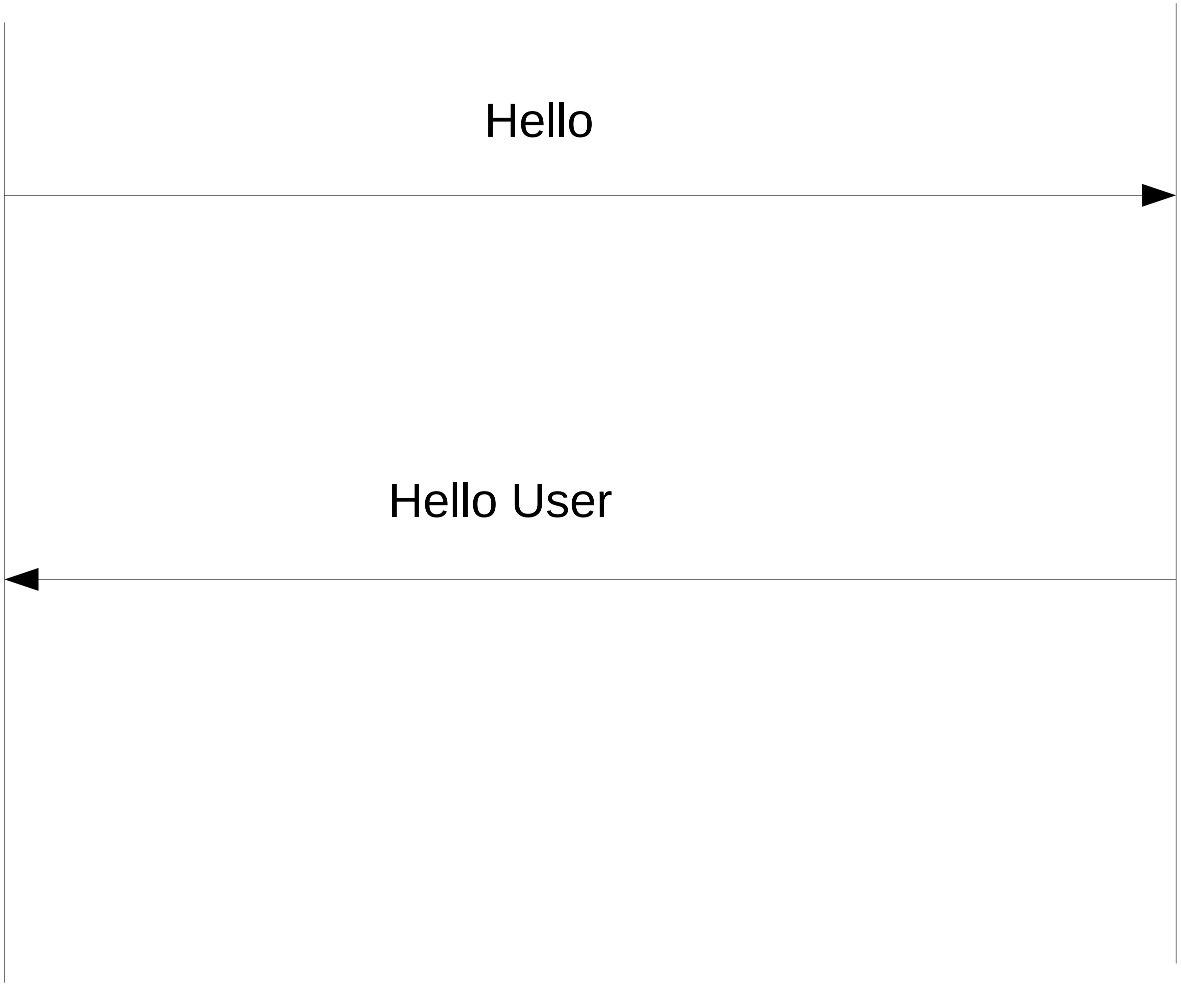
{“msg”:”login”, “name”:”joe”}

Bad instance

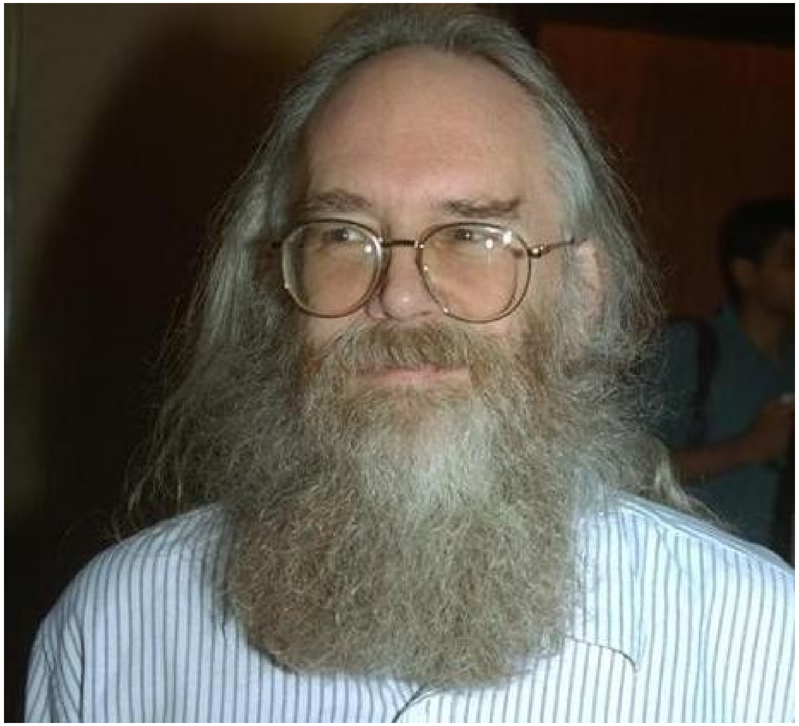
{“msg”:”login”, “footsize”:42}

Contracts and version bliss

Version hell



Postel's law



An implementation should be conservative in it's sending behavior, and liberal in its receiving behavior" (reworded from in RFC 1122 as "Be liberal in what you accept, and conservative in what you send")

"Making matters worse - law"

August 1982
Jonathan B. Postel

RFC 821

TCP Port 25

APPENDIX E

- Theory of Reply Codes
- 1yx Positive preliminary reply
- 2yz Positive Completion reply
- 3yz Positive Intermediate reply
- 4yz Transient Negative Completion Reply

Protocol

3.5. OPENING AND CLOSING

At the time the transmission channel is opened there is an exchange to ensure that the hosts are communicating with the hosts they think they are.

The following two commands are used in transmission channel opening and closing:

```
HELO <SP> <domain> <CRLF>

QUIT <CRLF>
```

In the HELO command the host sending the command identifies itself; the command may be interpreted as saying "Hello, I am <domain>".

Protocol

```
               Example of Connection Opening

         R: 220 BBN-UNIX.ARPA Simple Mail Transfer Service Ready
         S: HELO USC-ISIF.ARPA
         R: 250 BBN-UNIX.ARPA

                            Example 5

    -------------------------------------------------------------

               Example of Connection Closing

         S: QUIT
         R: 221 BBN-UNIX.ARPA Service closing transmission
```

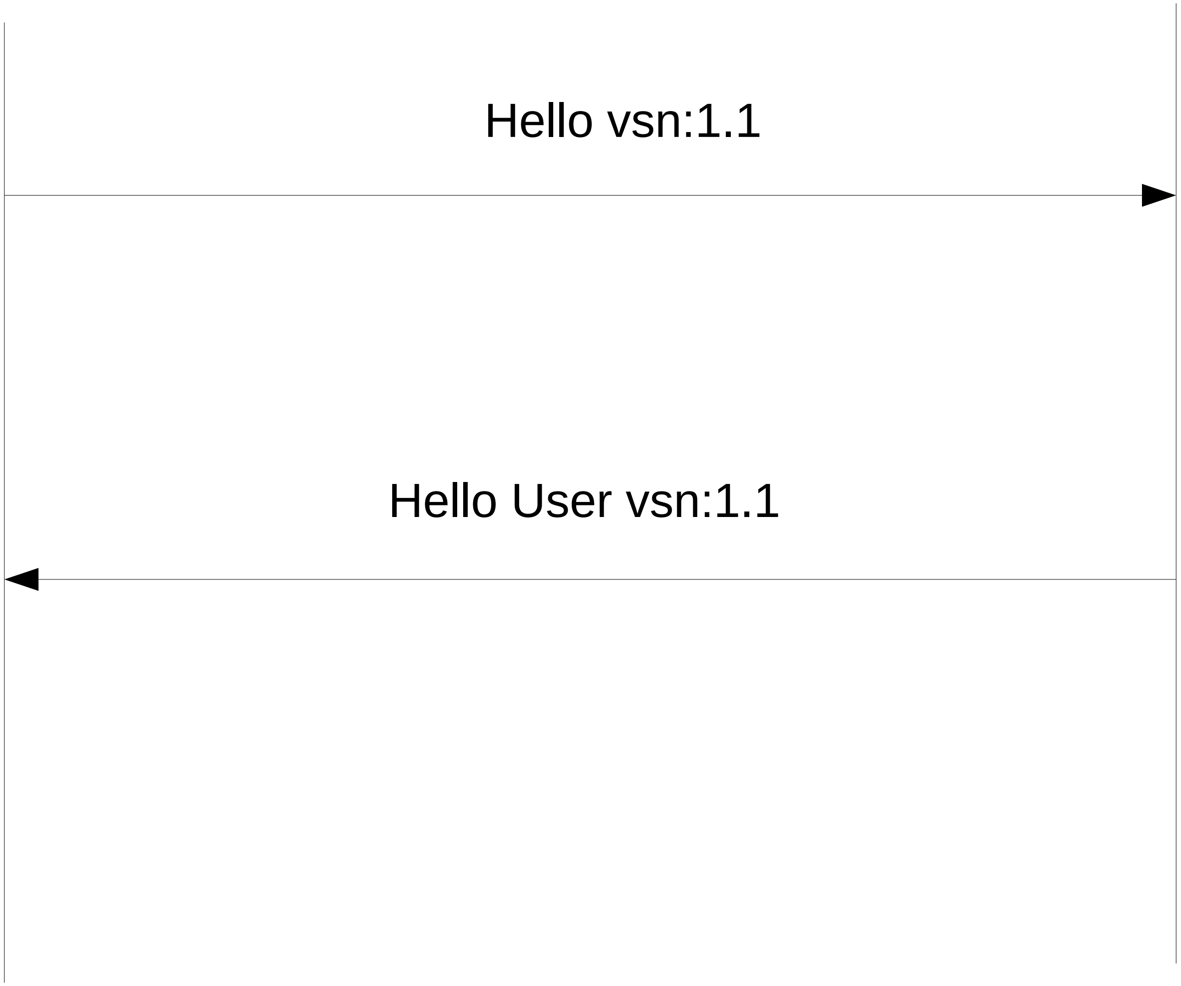
April 2001
RFC 2821
Simple Mail Transfer Protocol

3.1 Session Initiation

An SMTP session is initiated when a client opens a connection to a server and the server responds with an opening message.

SMTP server implementations MAY include identification of their software and version information in the connection greeting reply after the 220 code, a practice that permits more efficient isolation and repair of any problems. Implementations MAY make provision for SMTP servers to disable the software and version announcement where it causes security concerns. While some systems also identify their contact point for mail problems, this is not a substitute for maintaining the required "postmaster" address (see section 4.5.1).

Version Purgatory



Hello vsn:1.1

Hello User vsn:1.1

HTTP 0.9: the one line protocol
http://www.w3.org/Protocols/HTTP/AsImplemented.html

```
$> telnet google.com 80
Connected to 74.125.xxx.xxx

 GET /about/

 (hypertext response)
 (connection closed)
```

RFC 1945
Hypertext Transfer Protocol -- HTTP/1.0
May 1996

The version of an HTTP message is indicated by an HTTP-Version field in the first line of the message. If the protocol version is not specified, the recipient must assume that the message is in the simple HTTP/0.9 format.

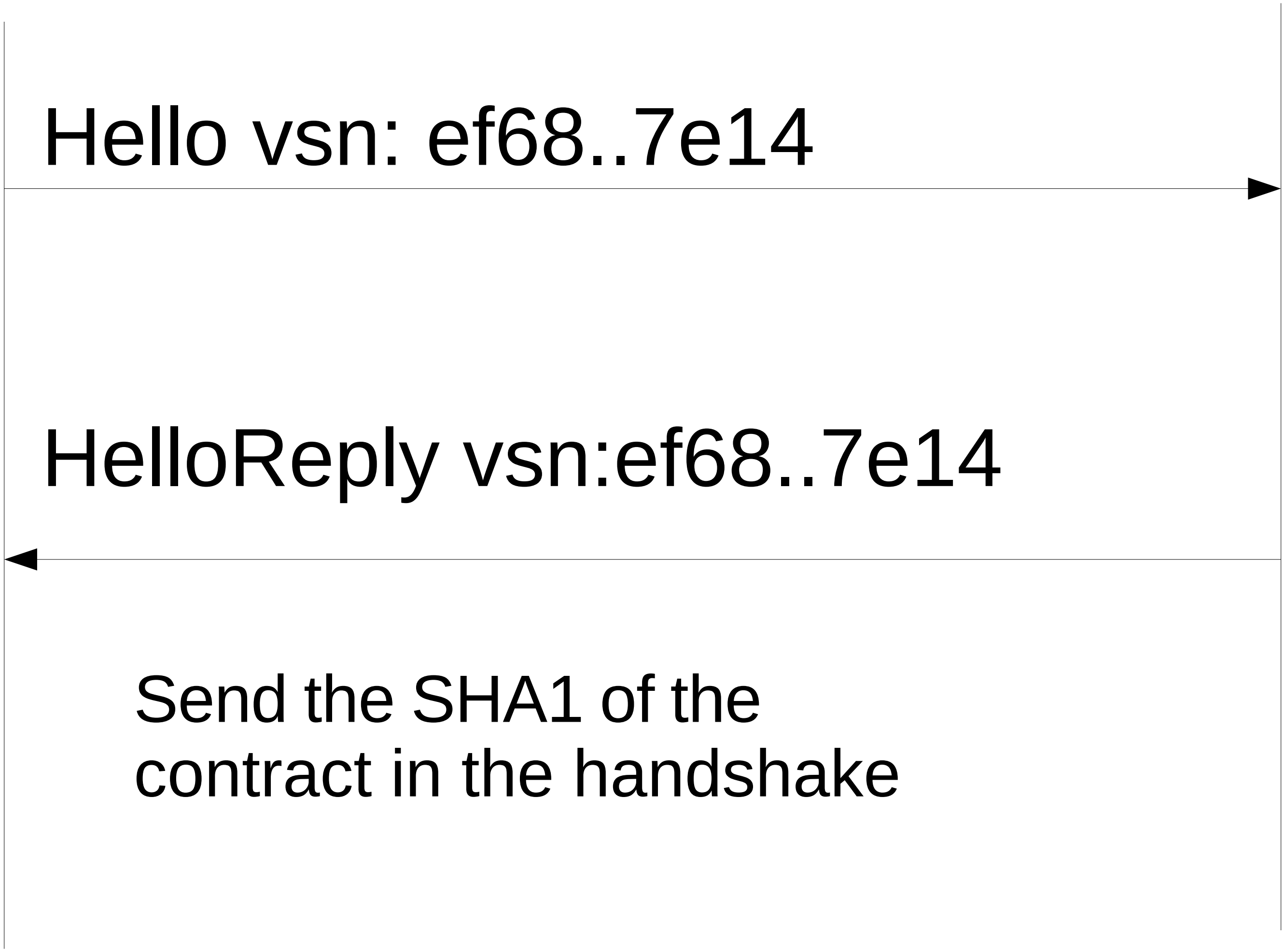
```
S: GET pageName HTTP/1.0

R: HTTP/1.0 200 OK
Date: Thu, 30 Oct 2008 18:17:16 GMT
```

It took 14 years to get the idea that version numbers in protocols might be a good idea

Content Hashes

- If two files are the same they have the same content hash (think md5, sha1, …)
- A directory can be described by a content hash (just hash the hashes of the individual files)
- An entire OS can be described by a single content hash (think sha1 of the iso) (NixOS)
- Protocols can be described by content hashes
- Data protected by content hashes is secure
- GIT :-)

Version Heaven?

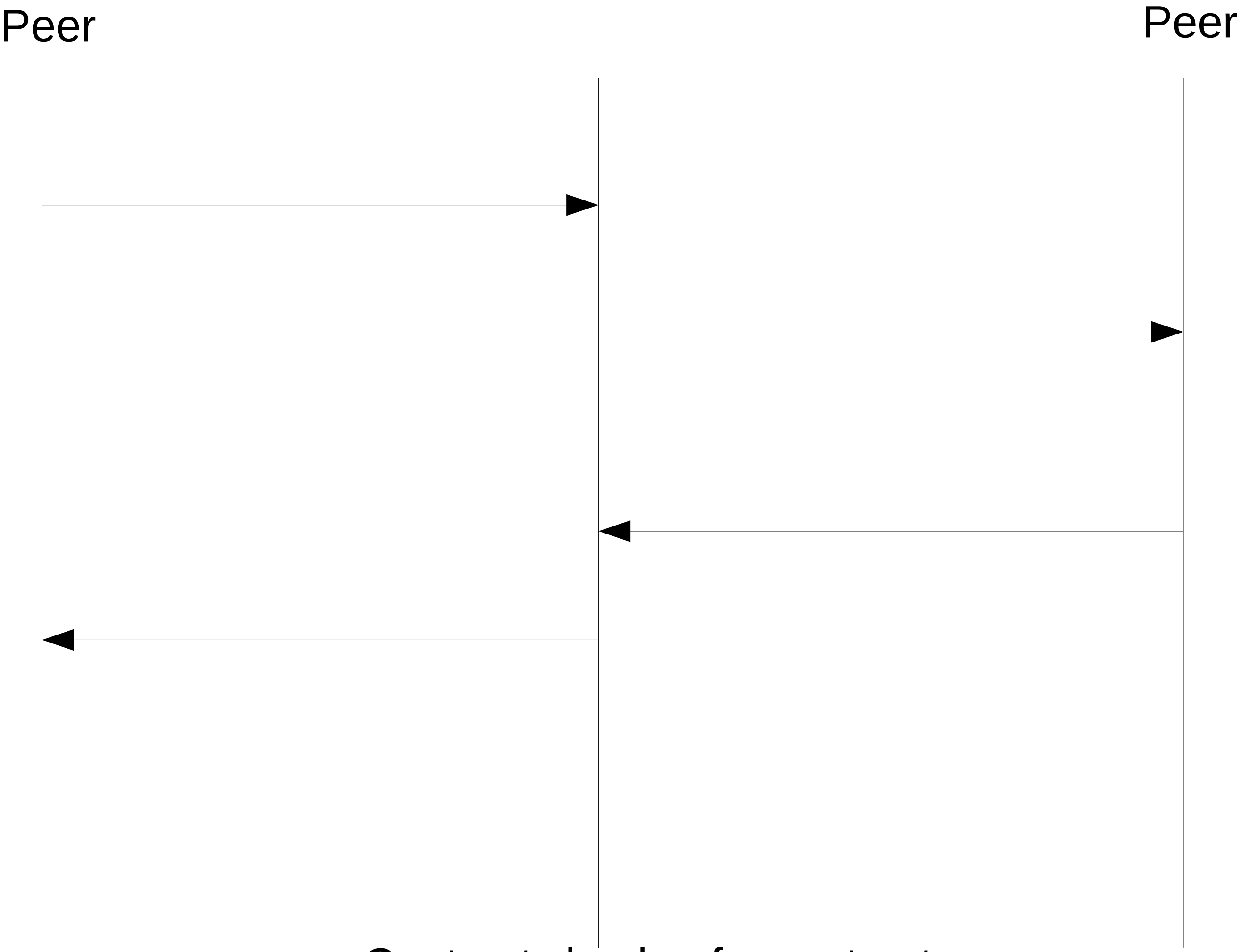


Hello vsn: ef68..7e14

HelloReply vsn:ef68..7e14

Send the SHA1 of the contract in the handshake

Contract heaven???

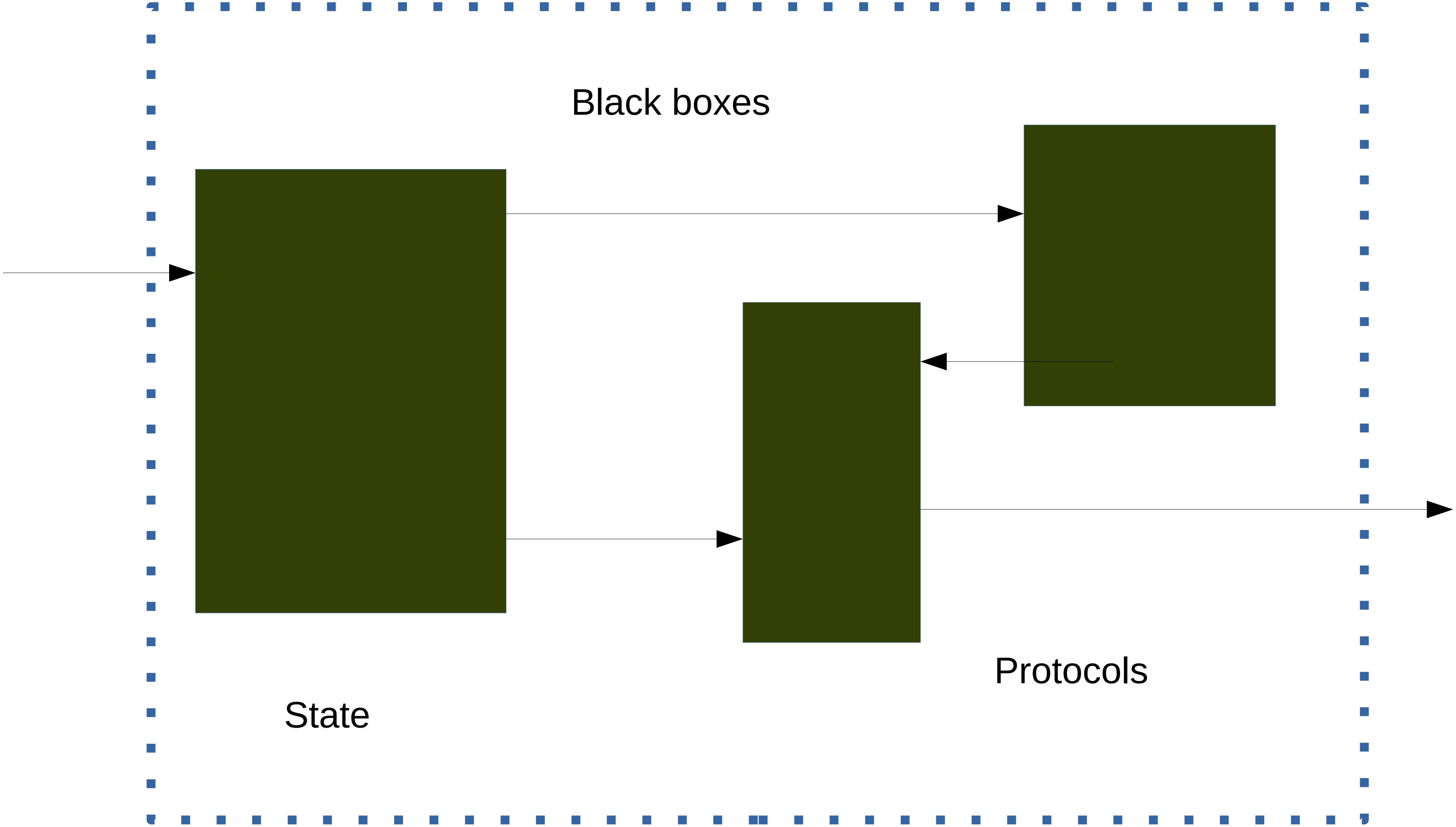


Peer

Peer

Contract checker for contract
ef68a891209121397922cbefb860d01864cbc7e1

The Big picture



Black boxes

State

Protocols

Abstraction

Questions